

# Introducing a Calculus of Effects and Handlers for Natural Language Semantics

Jirka Maršík and Maxime Amblard

LORIA, UMR 7503, Université de Lorraine, CNRS, Inria, Campus Scientifique,  
F-54506 Vandœuvre-lès-Nancy, France  
{jirka.marsik, maxime.amblard}@loria.fr

**Abstract.** In compositional model-theoretic semantics, researchers assemble truth-conditions or other kinds of denotations using the lambda calculus. It was previously observed [26] that the lambda terms and/or the denotations studied tend to follow the same pattern: they are instances of a monad. In this paper, we present an extension of the simply-typed lambda calculus that exploits this uniformity using the recently discovered technique of effect handlers [22]. We prove that our calculus exhibits some of the key formal properties of the lambda calculus and we use it to construct a modular semantics for a small fragment that involves multiple distinct semantic phenomena.

**Keywords:** compositionality, side effects, monads, handlers, deixis, conventional implicature

## 1 Introduction

The prevailing methodology of formal semantics is compositionality in the sense of Frege: denotations of complex phrases are functions of the denotations of their immediate constituents. However, several phenomena have been identified that challenge this notion of compositionality. Examples include anaphora, presupposition, quantification, deixis and conventional implicature. In all of these examples, simple models of denotation (i.e. noun phrases are individuals, sentences are truth-values) run into complications as the denotations can depend on external values (anaphora, deixis) or on something which is not an immediate constituent (presupposition, quantification, conventional implicature).

Among the solutions to these challenges, we find (at least) two types of solutions. First, we have those that relax the condition of compositionality. Notably, the denotation of a complex phrase is no longer a *function per se* of the denotations of its immediate subconstituents. Rather, it is some other formally defined process.<sup>1</sup> Examples of this approach include:

---

<sup>1</sup> This kind of distinction is the same distinction as the one between a mathematical function and a function in a programming language, which might have all kinds of side effects and therefore not be an actual function.

- the incremental algorithm used to build discourse representation structures in DRT, as presented in [12]
- the  $\lambda\mu$  calculus, used in [6] to analyze quantification, since, due to the lack of confluence, function terms do not denote functions over simple denotations
- the use of exceptions and exception handlers in [18] to model presuppositions in an otherwise compositional framework
- the parsetree interpretation step in the logic of conventional implicatures of [23] that builds the denotation of a sentence by extracting implicatures from the denotations of its subparts (including the non-immediate ones)

The other approach is to enrich the denotations so that they are parameterized by the external information they need to obtain and contain whatever internal information they need to provide to their superconstituents. Here are some examples of this style:

- any kind of semantic indices (e.g. the speaker and addressee for deixis, the current world for modality), since they amount to saying that a phrase denotes an indexed set of simpler meanings
- the continuized semantics for quantification [1] in which denotations are functions of their own continuations
  - and more generally, any semantics using type raising or generalized quantifiers for noun phrase denotations
- the dynamic denotations of [7] that are functions of the common ground and their continuation
- compositional event semantics, such as the one in [25], that shift the denotations of sentences from truth-values to predicates on events

We want to find a common language in which we could express the above techniques. Our inspiration comes from computer science. There, a concept known as *monad* has been used:

- in denotational semantics to give the domain of interpretation for programming languages that involve side effects [21].
- in functional programming to emulate programming with side effects via term-level encodings of effectful programs [29].

These two principal applications of monads align with the two approaches we have seen above. The one where we change our calculus so it no longer defines pure functions (e.g. is non-deterministic, stateful or throws exceptions) and the one where we use a pure calculus to manipulate terms (denotations) that encode some interaction (e.g. dynamicity, continuations or event predication).

Monad is a term from category-theory. Its meaning is relative to a category. For us, this will always be the category whose objects are types and whose arrows are functions between different types. A monad is formed by a functor and a pair of natural transformations that satisfy certain laws. In our case, this means that a monad is some type constructor (the functor part) and some combinators (the natural transformations) that follow some basic laws. To give an example of this,

we can think of the functor  $T(\alpha) = (\alpha \rightarrow o) \rightarrow o$  together with combinators such as the type raising  $\eta(x) = \lambda P. P x$  as a monad of quantification.

The relationship between side effects in functional programming and computational semantics has been developed in several works [27,28],<sup>2</sup> stretching as far back as 1977 [10]. The usefulness of monads in particular has been discovered by Shan in 2002 [26]. Since then, the problem that remained was how to compose several different monads in a single solution. Charlow used the popular method of monad morphisms<sup>3</sup> to combine several monads in his dissertation [4]. Giorgolo and Asudeh have used distributive laws to combine monads [8], while Kiselyov has eschewed monads altogether in favor of applicative functors which enjoy easy composability [13].

Our approach follows the recent trend in adopting effects and handlers to combine side effects [2,11] and to encode effectful programs in pure functional programming languages [14,3].

The idea is that we can represent each of the relevant monads using an algebra. We can then combine the signatures of the algebras by taking a disjoint union. The free algebra of the resulting signature will serve as a universal representation format for the set of all terms built from any of the source algebras and closed under substitution. Then, we will build modular interpreters that will give meanings to the operators of the algebras in terms of individuals, truth-values and functions.

In Sect. 2, we will introduce a formal calculus for working with the algebraic terms that we will use in our linguistic denotations. In Sect. 3, we will incrementally build up a fragment involving several of the linguistic phenomena and see the calculus in action. Before we conclude in Sect. 5, we will also discuss some of the formal properties of the calculus in Sect. 4.

## 2 Definition of the Calculus

Our calculus is an extension of the simply-typed lambda calculus (STLC). We add terms of a free algebra into our language and a notation for writing handlers, composable interpreters of these terms. An operator of the free algebra corresponds to a particular interaction that a piece of natural language can have with its context (e.g. a deictic expression might request the speaker's identity using some operator **speaker** in order to find its denotation). A handler gives an interpretation to every occurrence of an operator within a term (e.g. direct speech introduces a handler for the operator **speaker** that essentially rebinds the current speaker to some other entity).

---

<sup>2</sup> Side effects are to programming languages what pragmatics are to natural languages: they both study how expressions interact with the worlds of their users. It might then come as no surprise that phenomena such as anaphora, presupposition, deixis and conventional implicature yield a monadic description.

<sup>3</sup> Also known as monad transformers in functional programming.

Having sketched the general idea behind our calculus, we will now turn our attention to the specifics. We start by defining the syntactic constructions used to build the terms of our language.

## 2.1 Terms

First off, let  $\mathcal{X}$  be a set of variables,  $\Sigma$  a typed signature and  $\mathcal{E}$  a set of operation symbols. In the definition below, we will let  $M, N \dots$  range over terms,  $x, y, z \dots$  range over variables from  $\mathcal{X}$ ,  $c, d \dots$  range over the names of constants from  $\Sigma$  and  $\text{op}, \text{op}_i \dots$  range over the operation symbols in  $\mathcal{E}$ .

The terms of our language are composed of the following:

|  |               |
|--|---------------|
| $M, N ::= \lambda x. M$  | [abstraction] |
| $M N$  | [application] |
| $x$  | [variable]    |
| $c$  | [constant]    |
| $\text{op } M_p (\lambda x. M_c)$                                | [operation]   |
| $\eta M$   | [injection]   |
| $(\text{op}_1 : M_1, \dots, \text{op}_n : M_n, \eta : M_\eta) N$ | [handler]     |
| $\circ M$  | [extraction]  |
| $\mathcal{C} M$  | [exchange]    |

The first four constructions — abstraction, application, variables and constants — come directly from STLC with constants.

The next four deal with the algebraic expressions used to encode computations. Let us sketch the behaviors of these four kinds of expressions.

The operation ( $\text{op}$ ) and injection ( $\eta$ ) expressions will serve as the constructors for our algebraic expressions. Algebraic expressions are usually formed by operation symbols and then variables as atoms. Instead of variables, our algebraic expressions use terms from our calculus for atoms. The  $\eta$  constructor can thus take an ordinary term from our calculus and make it an atomic algebraic expression. The operation symbols  $\text{op}$  are then the operations of the algebra.

The other three expression types correspond to functions over algebraic expressions.

- The most useful is the handler  $(\text{op})$ .<sup>4</sup> It is an iterator for the type of algebraic expressions. The terms  $M_1, \dots, M_n$  and  $M_\eta$  in  $(\text{op}_1 : M_1, \dots, \text{op}_n : M_n, \eta : M_\eta)$  are the clauses for the constructors  $\text{op}_1, \dots, \text{op}_n$  and  $\eta$ , respectively. We will use handlers to define interpretations of operation symbols in algebraic expressions.
- The cherry  $\circ$  operator allows us to extract terms out of algebraic expressions. If an algebraic expression is of the form  $\eta M$ , applying  $\circ$  to it will yield  $M$ .

<sup>4</sup> Pronounced “banana”. See [20] for the introduction of banana brackets.

- The exchange operator  $\mathcal{C}$  permits a kind of commutation between the  $\lambda$ -binder and the operation symbols. We will see its use later.

## 2.2 Types

We now give a syntax for the types of our calculus along with a typing relation. In the grammar below,  $\alpha, \beta, \gamma \dots$  range over types,  $\nu$  ranges over atomic types from some set  $\mathcal{T}$  and  $E, E' \dots$  range over effect signatures (introduced below).

The types of our language consist of:

$$\begin{array}{ll} \alpha, \beta, \gamma ::= \alpha \rightarrow \beta & \text{[function]} \\ \quad \mid \nu & \text{[atom]} \\ \quad \mid \mathcal{F}_E(\alpha) & \text{[computation]} \end{array}$$

The only novelty here is the  $\mathcal{F}_E(\alpha)$  computation type. This is the type of algebraic expressions whose atoms are terms of type  $\alpha$  and whose operation symbols come from the effect signature  $E$ . We call them *computation types* and we call terms of these types *computations* because our algebraic expressions will always represent some kind of program with effects.

*Effect signatures* are similar to typing contexts. They are partial mappings from the set of operation symbols  $\mathcal{E}$  to pairs of types. We will write the elements of effect signatures the following way —  $\text{op} : \alpha \mapsto \beta \in E$  means that  $E$  maps  $\text{op}$  to the pair of types  $\alpha$  and  $\beta$ .<sup>5</sup> When dealing with effect signatures, we will often make use of the disjoint union operator  $\uplus$ . The term  $E_1 \uplus E_2$  serves as a constraint demanding that the domains of  $E_1$  and  $E_2$  be disjoint and at the same time it denotes the effect signature that is the union of  $E_1$  and  $E_2$ .

The typing rules are presented in Figure 1.

The typing rules mirror the syntax of terms. Again, the first four rules come from STLC. The  $[\eta]$  and  $[\downarrow]$  rules are self-explanatory and so we will focus on the  $[\text{op}]$ ,  $[\llbracket \cdot \rrbracket]$  and  $[\mathcal{C}]$  rules.

**[op]** To use an operation  $\text{op} : \alpha \mapsto \beta$ , we provide the input parameter  $M_p : \alpha$  and a continuation  $\lambda x. M_c : \beta \rightarrow \mathcal{F}_E(\gamma)$ , which expects the output of type  $\beta$ . The resulting term has the same type as the body of the continuation,  $\mathcal{F}_E(\gamma)$ .

Before, we have spoken of terms of type  $\mathcal{F}_E(\gamma)$  as of algebraic expressions generated by the terms of type  $\gamma$  and the operators in the effect signature  $E$ . However, having seen the typing rule for operation terms, it might not be obvious how such a term represents an algebraic expression. Traditionally, algebraic signatures map operation symbols to arities, which are natural numbers. Our effect signatures map each operation symbol to a pair of types  $\alpha \mapsto \beta$ .

- We can explain  $\alpha$  by analogy to the single-sorted algebra of vector spaces.

In a single-sorted vector space algebra, scalar multiplication is viewed as

---

<sup>5</sup> The two types  $\alpha$  and  $\beta$  are to be seen as the operation's *input* and *output* types, respectively.

$$\begin{array}{c}
\frac{\Gamma, x : \alpha \vdash M : \beta}{\Gamma \vdash \lambda x. M : \alpha \rightarrow \beta} [\text{abs}] \qquad \frac{\Gamma \vdash M : \alpha \rightarrow \beta \quad \Gamma \vdash N : \alpha}{\Gamma \vdash MN : \beta} [\text{app}] \\
\\
\frac{x : \alpha \in \Gamma}{\Gamma \vdash x : \alpha} [\text{var}] \qquad \frac{c : \alpha \in \Sigma}{\Gamma \vdash c : \alpha} [\text{const}] \\
\\
\frac{\Gamma \vdash M : \alpha}{\Gamma \vdash \eta M : \mathcal{F}_E(\alpha)} [\eta] \qquad \frac{\Gamma \vdash M_p : \alpha \quad \Gamma, x : \beta \vdash M_c : \mathcal{F}_E(\gamma) \quad \text{op} : \alpha \multimap \beta \in E}{\Gamma \vdash \text{op } M_p (\lambda x. M_c) : \mathcal{F}_E(\gamma)} [\text{op}] \\
\\
\frac{\Gamma \vdash M : \mathcal{F}_\emptyset(\alpha)}{\Gamma \vdash \circ M : \alpha} [\circ] \qquad \frac{E = \{\text{op}_i : \alpha_i \multimap \beta_i\}_{i \in I} \uplus E_f \quad E' = E'' \uplus E_f \quad [\Gamma \vdash M_i : \alpha_i \rightarrow (\beta_i \rightarrow \mathcal{F}_{E'}(\delta)) \rightarrow \mathcal{F}_{E'}(\delta)]_{i \in I} \quad \Gamma \vdash M_\eta : \gamma \rightarrow \mathcal{F}_{E'}(\delta) \quad \Gamma \vdash N : \mathcal{F}_E(\gamma)}{\Gamma \vdash \llbracket (\text{op}_i : M_i)_{i \in I}, \eta : M_\eta \rrbracket N : \mathcal{F}_{E'}(\delta) \rrbracket} [\llbracket \rrbracket] \\
\\
\frac{\Gamma \vdash M : \alpha \rightarrow \mathcal{F}_E(\beta)}{\Gamma \vdash \mathcal{C} M : \mathcal{F}_E(\alpha \rightarrow \beta)} [\mathcal{C}]
\end{array}$$

Fig. 1: The typing rules for our calculus.

a unary operation parameterized by some scalar. So technically, there is a different unary operation for each scalar. All of our operations are similarly parameterized and  $\alpha$  is the type of that parameter.

- The type  $\beta$  expresses the arity of the operator. When we say that an operator has arity  $\beta$ , where  $\beta$  is a type, we mean that it takes one operand for every value of  $\beta$  [24]. We can also think of the operator as taking one operand containing  $x : \beta$  as a free variable.

We can look at the algebraic expression  $\text{op } M_p (\lambda x. M_c)$  as a description of a program that:

- interacts with its context by some operator called  $\text{op}$
- to which it provides the input  $M_p$
- and from which it expects to receive an output of type  $\beta$
- which it will then bind as the variable  $x$  and continue as the program described by  $M_c$ .

$\llbracket \rrbracket$  The banana brackets describe iterators/catamorphisms.<sup>6</sup> In the typing rule,  $E$  is the input's signature,  $E'$  is the output's signature,  $\gamma$  is the input's atom type and  $\delta$  is the output's atom type.  $E$  is decomposed into the operations that our

<sup>6</sup> These are similar to recursors/paramorphisms. See [20] for the difference. Catamorphisms are also known as folds and the common higher-order function *fold* found in functional programming languages is actually the iterator/catamorphism for lists.

iterator will actually interpret, the other operations form a residual signature  $E_f$ . The output signature will then still contain the uninterpreted operations  $E_f$  combined with any operations  $E''$  that our interpretation might introduce.

[**C**] We said before that the  $\mathcal{C}$  function will let us commute  $\lambda$  and operations. Here we see that, on the type level, this corresponds to commuting the  $\mathcal{F}_E(-)$  and the  $\alpha \rightarrow \_$  type constructors.

### 2.3 Reduction Rules

We will now finally give a semantics to our calculus. The semantics will be given in the form of a reduction relation on terms. Even though the point of the calculus is to talk about effects, the reduction semantics will not be based on any fixed evaluation order; any subterm that is a redex can be reduced in any context. The reduction rules are given in Fig. 2.

|  |  |
|--|--|
| $(\lambda x. M) N \rightarrow$<br>$M[x := N]$  | rule $\beta$   |
| $\lambda x. M x \rightarrow$<br>$M$  | rule $\eta$<br>where $x \notin \text{FV}(M)$   |
| $\llbracket (\text{op}_i; M_i)_{i \in I}, \eta: M_\eta \rrbracket (\eta N) \rightarrow$<br>$M_\eta N$  | rule $\llbracket \eta \rrbracket$  |
| $\llbracket (\text{op}_i; M_i)_{i \in I}, \eta: M_\eta \rrbracket (\text{op}_j N_p (\lambda x. N_c)) \rightarrow$<br>$M_j N_p (\lambda x. \llbracket (\text{op}_i; M_i)_{i \in I}, \eta: M_\eta \rrbracket N_c)$         | rule $\llbracket \text{op} \rrbracket$<br>where $j \in I$<br>and $x \notin \text{FV}((M_i)_{i \in I}, M_\eta)$     |
| $\llbracket (\text{op}_i; M_i)_{i \in I}, \eta: M_\eta \rrbracket (\text{op}_j N_p (\lambda x. N_c)) \rightarrow$<br>$\text{op}_j N_p (\lambda x. \llbracket (\text{op}_i; M_i)_{i \in I}, \eta: M_\eta \rrbracket N_c)$ | rule $\llbracket \text{op}' \rrbracket$<br>where $j \notin I$<br>and $x \notin \text{FV}((M_i)_{i \in I}, M_\eta)$ |
| $\downarrow (\eta M) \rightarrow$<br>$M$   | rule $\downarrow$  |
| $\mathcal{C} (\lambda x. \eta M) \rightarrow$<br>$\eta (\lambda x. M)$   | rule $\mathcal{C}_\eta$  |
| $\mathcal{C} (\lambda x. \text{op } M_p (\lambda y. M_c)) \rightarrow$<br>$\text{op } M_p (\lambda y. \mathcal{C} (\lambda x. M_c))$   | rule $\mathcal{C}_{\text{op}}$<br>where $x \notin \text{FV}(M_p)$  |

Fig. 2: The reduction rules of our calculus.

We have the  $\beta$  and  $\eta$  rules, which, by no coincidence, are the same rules as the ones found in STLC. The rest are function definitions for  $\llbracket \_ \rrbracket$ ,  $\downarrow$  and  $\mathcal{C}$ .

By looking at the definition of  $\llbracket \cdot \rrbracket$ , we see that it is an iterator. It replaces every occurrence of the constructors  $\text{op}_j$  and  $\eta$  with  $M_j$  and  $M_\eta$ , respectively.

The  $\mathcal{C}$  function recursively swaps  $\mathcal{C}(\lambda x. \_)$  with  $\text{op } M_p(\lambda y. \_)$  using the  $\mathcal{C}_{\text{op}}$  rule. When  $\mathcal{C}$  finally meets the  $\eta$  constructor, it swaps  $(\lambda x. \_)$  with  $\eta \_$  and terminates. Note that the constraint  $x \notin \text{FV}(M_p)$  in rule  $\mathcal{C}_{\text{op}}$  cannot be dismissed by renaming of bound variables. If the parameter  $M_p$  contains a free occurrence of  $x$ , the evaluation of  $\mathcal{C}$  will get stuck.  $\mathcal{C}$  is thus a partial function: it is only applicable when none of the operations being commuted with the  $\lambda$ -binder actually depend on the bound variable.

## 2.4 Common Combinators

When demonstrating the calculus in the next section, the following combinators will be helpful. First, we define a sequencing operator. The operator  $\gg=$ , called bind, replaces all the  $\alpha$ -typed atoms of a  $\mathcal{F}_E(\alpha)$ -typed expression with  $\mathcal{F}_E(\beta)$ -typed expressions. More intuitively,  $M \gg= N$  is the program that first runs  $M$  to get its result  $x$  and then continues as the program  $N x$ .

$$\begin{aligned} \_ \gg= \_ &: \mathcal{F}_E(\alpha) \rightarrow (\alpha \rightarrow \mathcal{F}_E(\beta)) \rightarrow \mathcal{F}_E(\beta) \\ M \gg= N &= \llbracket \eta: N \rrbracket M \end{aligned}$$

The type constructor  $\mathcal{F}_E$  along with the operators  $\eta$  and  $\gg=$  form a free monad. Using this monadic structure, we can define the following combinators (variations on application) which we will make heavy use of in Section 3.

$$\begin{aligned} \_ \ll \cdot \_ &: \mathcal{F}_E(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \mathcal{F}_E(\beta) \\ F \ll \cdot x = F \gg= &(\lambda f. \eta(f x)) \\ \_ \cdot \gg \_ &: (\alpha \rightarrow \beta) \rightarrow \mathcal{F}_E(\alpha) \rightarrow \mathcal{F}_E(\beta) \\ f \cdot \gg X = X \gg= &(\lambda x. \eta(f x)) \\ \_ \ll \cdot \gg \_ &: \mathcal{F}_E(\alpha \rightarrow \beta) \rightarrow \mathcal{F}_E(\alpha) \rightarrow \mathcal{F}_E(\beta) \\ F \ll \cdot \gg X = F \gg= &(\lambda f. X \gg= (\lambda x. \eta(f x))) \end{aligned}$$

All of these operators associate to the left, so  $f \cdot \gg X \ll \cdot \gg Y$  should be read as  $(f \cdot \gg X) \ll \cdot \gg Y$ .

Let  $\circ : o \rightarrow o \rightarrow o$  be a binary operator on propositions. We define the following syntax for the same operator lifted to computations of propositions.

$$\begin{aligned} \_ \bar{\circ} \_ &: \mathcal{F}_E(o) \rightarrow \mathcal{F}_E(o) \rightarrow \mathcal{F}_E(o) \\ M \bar{\circ} N &= (\lambda mn. m \circ n) \cdot \gg M \ll \cdot \gg N \end{aligned}$$

## 3 Linguistic Phenomena as Effects

### 3.1 Deixis

We will now try to use this calculus to do some semantics. Here is our tectogrammar in an abstract categorial grammar presentation [5].



$$\begin{aligned} \text{JOHN, MARY, ME} &: NP \\ \text{LOVES} &: NP \multimap NP \multimap S \end{aligned}$$

And here is our semantics.

$$\begin{aligned} \llbracket \text{JOHN} \rrbracket &:= \eta \mathbf{j} \\ \llbracket \text{MARY} \rrbracket &:= \eta \mathbf{m} \\ \llbracket \text{ME} \rrbracket &:= \mathbf{speaker} \star (\lambda x. \eta x) \\ \llbracket \text{LOVES} \rrbracket &:= \lambda OS. \mathbf{love} \cdot \gg S \ll \gg O \end{aligned}$$

In the semantics for  $\llbracket \text{ME} \rrbracket$ , we use the **speaker** operation to retrieve the current speaker and make it available as the value of the variable  $x$ . The star ( $\star$ ) passed to **speaker** is a dummy value of the unit type 1.

This, and all the semantics we will see in this paper, satisfies a homomorphism condition that whenever  $M : \tau$ , then  $\llbracket M \rrbracket : \llbracket \tau \rrbracket$ . In our case,  $\llbracket NP \rrbracket = \mathcal{F}_E(\iota)$  and  $\llbracket S \rrbracket = \mathcal{F}_E(o)$ , where  $\iota$  and  $o$  are the types of individuals and propositions, respectively. Of  $E$ , we assume that  $\mathbf{speaker} : 1 \multimap \iota \in E$ , since that is the type of **speaker** used in our semantics.<sup>7</sup>

With this fragment, we can give meanings to trivial sentences like:

- (1) John loves Mary.
- (2) Mary loves me.

whose meanings we can calculate as:

$$\llbracket \text{LOVES MARY JOHN} \rrbracket \multimap \eta (\mathbf{love} \mathbf{j} \mathbf{m}) \tag{1}$$

$$\llbracket \text{LOVES ME MARY} \rrbracket \multimap \mathbf{speaker} \star (\lambda x. \eta (\mathbf{love} \mathbf{m} x)) \tag{2}$$

The meaning of (1) is a proposition of type  $o$  wrapped in  $\eta$ , i.e. something that we can interpret in a model. As for the meaning of (2), the **speaker** operator has propagated from the ME lexical entry up to the meaning of the whole sentence. We now have an algebraic expression having as operands the propositions  $\mathbf{love} \mathbf{m} x$  for all possible  $x : \iota$ . In order to get a single proposition which is to be seen as the truth-conditional meaning of the sentence and which can be evaluated in a model, we will need to fix the speaker. We will do so by defining an interpreting handler.

$$\begin{aligned} \text{withSpeaker} &: \iota \rightarrow \mathcal{F}_{\{\mathbf{speaker}: 1 \multimap \iota\} \uplus E}(\alpha) \rightarrow \mathcal{F}_E(\alpha) \\ \text{withSpeaker} &= \lambda s M. (\llbracket \mathbf{speaker} \rrbracket (\lambda x k. k s)) M \end{aligned}$$

Note that we omitted the  $\eta$  clause in the banana brackets above. In such cases, we say there is a default clause  $\eta: (\lambda x. \eta x)$ .

---

<sup>7</sup> 1 is the unit type whose only element is written as  $\star$ .

$$\text{withSpeaker } s \llbracket \text{LOVES ME MARY} \rrbracket \rightarrow \eta (\text{love } \mathbf{m} \ s)$$

So far, we could have done the same by introducing a constant named **me** to stand in for the speaker. However, since handlers are part of our object language, we can include them in lexical entries. With this, we can handle phenomena such as direct (quoted) speech, that rebinds the current speaker in a certain scope.

$$\text{SAID}_{\text{IS}} : S \multimap NP \multimap S$$

$$\text{SAID}_{\text{DS}} : S \multimap NP \multimap S$$

Those are our new syntactic constructors: one for the indirect speech use of *said* and the other for the direct speech use (their surface realizations would differ typographically or phonologically). Let us give them some semantics.

$$\begin{aligned} \llbracket \text{SAID}_{\text{IS}} \rrbracket &= \lambda CS. \mathbf{say} \cdot \gg S \ll \gg C \\ &= \lambda CS. S \gg = (\lambda s. \mathbf{say} \ s \cdot \gg C) \\ \llbracket \text{SAID}_{\text{DS}} \rrbracket &= \lambda CS. S \gg = (\lambda s. \mathbf{say} \ s \cdot \gg (\text{withSpeaker } s \ C)) \end{aligned}$$

Here we elaborated the entry for indirect speech so it is easier to compare with the one for direct speech, which just adds a use of the *withSpeaker* operator.

(3) John said Mary loves me.

(4) John said, “Mary loves me”.

$$\llbracket \text{SAID}_{\text{IS}} (\text{LOVES ME MARY}) \text{ JOHN} \rrbracket \rightarrow \mathbf{speaker} \star (\lambda x. \eta (\mathbf{say} \ \mathbf{j} (\text{love } \mathbf{m} \ x))) \quad (3)$$

$$\llbracket \text{SAID}_{\text{DS}} (\text{LOVES ME MARY}) \text{ JOHN} \rrbracket \rightarrow \eta (\mathbf{say} \ \mathbf{j} (\text{love } \mathbf{m} \ \mathbf{j})) \quad (4)$$

The meaning of sentence (3) depends on the speaker (as testified by the use of the **speaker** operator) whereas in (4), this dependence has been eliminated due to the use of direct speech.

### 3.2 Quantification

Now we turn our attention to quantificational noun phrases.

$$\text{EVERY, A} : N \multimap NP$$

$$\text{MAN, WOMAN} : N$$

$$\llbracket \text{EVERY} \rrbracket := \lambda N. \mathbf{scope} (\lambda c. \forall \cdot \gg (\mathcal{C} (\lambda x. (N \ll \cdot x) \Rightarrow (cx)))) (\lambda x. \eta \ x)$$

$$\llbracket \text{A} \rrbracket := \lambda N. \mathbf{scope} (\lambda c. \exists \cdot \gg (\mathcal{C} (\lambda x. (N \ll \cdot x) \overline{\wedge} (cx)))) (\lambda x. \eta \ x)$$

$$\llbracket \text{MAN} \rrbracket := \eta \ \mathbf{man}$$

$$\llbracket \text{WOMAN} \rrbracket := \eta \ \mathbf{woman}$$

The entries for **EVERY** and **A** might seem intimidating. However, if we ignore the  $\cdot \gg$ , the  $\mathcal{C}$ , the  $\ll \cdot$  and the overline on the logical operator, we get the familiar

generalized quantifiers. These decorations are the plumbing that takes care of the proper sequencing of effects.

Note that we make use of the  $\mathcal{C}$  operator here. In the denotation of  $\llbracket A \rrbracket$ , the term  $(\lambda x. (N \ll x) \overline{\wedge} (cx))$  describes the property to which we want to apply the quantifier  $\exists$ . However, this term is of type  $\iota \rightarrow \mathcal{F}_E(o)$ . In order to apply  $\exists$ , we need something of type  $\iota \rightarrow o$ . Intuitively, the effects of  $E$  correspond to the process of interpretation, the process of arriving at some logical form of the sentence. They should thus be independent of the particular individual that we use as a witness for  $x$  when we try to model-check the resulting logical form. This independence allows us use the  $\mathcal{C}$  operator without fear of getting stuck. Once we arrive at the type  $\mathcal{F}_E(\iota \rightarrow o)$ , it is a simple case of using  $\exists \cdot \gg \_$  to apply the quantifier within the computation type.<sup>89</sup>

While the terms that use the `scope` operator might be complex, the handler that interprets them is as simple as can be.

$$\text{SI} = \lambda M. \langle \text{scope}: (\lambda ck. ck) \rangle M$$

Same as with `withSpeaker`, `SI` will also be used in lexical items. By interpreting the `scope` operation in a particular place, we effectively determine the scope of the quantifier. Hence the name of `SI`, short for `Scope Island`. If we want to model clause boundaries as scope islands, we can do so by inserting `SI` in the lexical entries of clause constructors (in our case, the verbs).

$$\begin{aligned} \llbracket \text{LOVES} \rrbracket &:= \lambda OS. \text{SI} (\llbracket \text{LOVES} \rrbracket O S) \\ \llbracket \text{SAID}_{\text{IS}} \rrbracket &:= \lambda CS. \text{SI} (\llbracket \text{SAID}_{\text{IS}} \rrbracket C S) \\ \llbracket \text{SAID}_{\text{DS}} \rrbracket &:= \lambda CS. \text{SI} (\llbracket \text{SAID}_{\text{DS}} \rrbracket C S) \end{aligned}$$

Whenever we use the semantic brackets on the right-hand side of these revised definitions, they stand for the denotations we have assigned previously.

- (5) Every man loves a woman.
- (6) John said every woman loves me.
- (7) John said, “Every woman loves me”.

$$\begin{aligned} &\llbracket \text{LOVES (A WOMAN) (EVERY MAN)} \rrbracket \\ &\rightarrow \eta (\forall x. \mathbf{man} \ x \rightarrow (\exists y. \mathbf{woman} \ y \wedge \mathbf{love} \ x \ y)) \end{aligned} \tag{5}$$

$$\begin{aligned} &\text{withSpeaker } s \llbracket \text{SAID}_{\text{IS}} (\text{LOVES ME (EVERY WOMAN)}) \text{ JOHN} \rrbracket \\ &\rightarrow \eta (\mathbf{say} \ \mathbf{j} (\forall x. \mathbf{woman} \ x \rightarrow \mathbf{love} \ x \ s)) \end{aligned} \tag{6}$$

$$\begin{aligned} &\llbracket \text{SAID}_{\text{DS}} (\text{LOVES ME (EVERY WOMAN)}) \text{ JOHN} \rrbracket \\ &\rightarrow \eta (\mathbf{say} \ \mathbf{j} (\forall x. \mathbf{woman} \ x \rightarrow \mathbf{love} \ x \ \mathbf{j})) \end{aligned} \tag{7}$$

<sup>8</sup> Other solutions to this problem include separating the language of logical forms and the metalanguage used in the semantic lexical entries to manipulate logical forms as objects [13].

<sup>9</sup> Our  $\mathcal{C}$  has been inspired by an operator of the same name proposed in [9]: de Groote introduces a structure that specializes applicative functors in a similar direction as monads by introducing the  $\mathcal{C}$  operator and equipping it with certain laws; our  $\mathcal{C}$  operator makes the  $\mathcal{F}_E$  type constructor an instance of this structure.

The calculus offers us flexibility when modelling the semantics. We might choose to relax the constraint that clauses are scope islands by keeping the old entries for verbs that do not use the SI handler. We might then want to add the SI handler to the lexical entry of  $\text{SAID}_{\text{DS}}$ , next to the  $\text{withSpeaker}$  handler, so that quantifiers cannot escape quoted expressions. We might also allow for inverse scope readings by, e.g., providing entries for transitive verbs that evaluate their arguments right-to-left (though then we would have to watch out for crossover effects if we were to add anaphora).

### 3.3 Conventional Implicature

Our goal is to show the modularity of this approach and so we will continue and plug in one more phenomenon into our growing fragment: conventional implicatures, as analyzed by Potts [23]. Specifically, we will focus on nominal appositives.

$$\begin{aligned} \text{APPOS} &: NP \multimap NP \multimap NP \\ \text{BEST-FRIEND} &: NP \multimap NP \\ \llbracket \text{APPOS} \rrbracket &:= \lambda XY. X \gg= (\lambda x. \text{SI}(\eta x \equiv Y) \gg= (\lambda i. \text{implicate } i (\lambda z. \eta x))) \\ \llbracket \text{BEST-FRIEND} \rrbracket &:= \lambda X. \text{best-friend} \cdot \gg X \end{aligned}$$

In the denotation of the nominal appositive construction,  $\text{APPOS}$ , we first evaluate the head noun phrase  $X : \llbracket NP \rrbracket$  to find its referent  $x : \iota$ . We then want to implicate that  $x$  is equal to the referent of  $Y$ . The term  $\eta x \equiv Y$  (note the line over  $=$ ) is the term that computes that referent and gives us the proposition we want. We also want to state that no quantifier from within the appositive  $Y$  should escape into the matrix clause and so we wrap this computation in the SI handler to establish a scope island. Finally, we pass this proposition as an argument to  $\text{implicate}$  and we return  $x$  as the referent of the noun phrase.

The point of the  $\text{implicate}$  operation is to smuggle non-at-issue content outside the scope of logical operators. The contribution of an appositive should survive, e.g., logical negation.<sup>10</sup> The place where we will accommodate the implicated truth-conditions will be determined by the use of the following handler:

$$\begin{aligned} \text{accommodate} &: \mathcal{F}_{\{\text{implicate}: o \rightarrow 1\} \uplus E}(o) \rightarrow \mathcal{F}_E(o) \\ \text{accommodate} &= \lambda M. (\llbracket \text{implicate}: (\lambda i k. \eta i \overline{\wedge} k \star) \rrbracket) M \end{aligned}$$

We want conventional implicatures to project out of the common logical operators. However, when we consider direct quotes, we would not like to attribute the implicature made by the quotee to the quoter. We can implement this by inserting the  $\text{accommodate}$  handler into the lexical entry for direct speech.

$$\llbracket \text{SAID}_{\text{DS}} \rrbracket := \lambda C S. \text{SI}(S \gg= (\lambda s. \text{say } s \cdot \gg (\text{withSpeaker } s (\text{accommodate } C))))$$

Consider the following three examples.

<sup>10</sup> In our limited fragment, we will only see it sneak out of a quantifier.

- (8) John, my best friend, loves every woman.
- (9) Mary, everyone’s best friend, loves John.
- (10) A man said, “My best friend, Mary, loves me”.

In (8), the conventional implicature that John is the speaker’s best friend projects from the scope of the quantifier. On the other hand, in (10), the implicature does not project from the quoted clause and so it is not misattributed.

$$\text{withSpeaker } s (\text{accommodate } \llbracket \text{LOVES (EVERY WOMAN) (APPOS JOHN (BEST-FRIEND ME))} \rrbracket) \\ \rightarrow \eta ((\mathbf{j} = \text{best-friend } s) \wedge (\forall x. \text{woman } x \rightarrow \text{love } \mathbf{j} x)) \quad (8)$$

$$\text{accommodate } \llbracket \text{LOVES JOHN (APPOS MARY (BEST-FRIEND EVERYONE))} \rrbracket \\ \rightarrow \eta ((\forall x. \mathbf{m} = \text{best-friend } x) \wedge (\text{love } \mathbf{m} \mathbf{j})) \quad (9)$$

$$\llbracket \text{SAID}_{\text{DS}} (\text{LOVES ME (APPOS (BEST-FRIEND ME) MARY)) (A MAN)} \rrbracket \\ \rightarrow \eta (\exists x. \text{man } x \wedge \text{say } x ((\text{best-friend } x = \mathbf{m}) \wedge (\text{love } (\text{best-friend } x) x))) \quad (10)$$

### 3.4 Summary

Let us look back at the modularity of our approach and count how often during the incremental development of our fragment we either had to modify existing denotations or explicitly mention previous effects in new denotations.

When adding quantification:

- in the old denotations of verbs, we added the new SI handler so that clauses form scope islands

When adding appositives and their conventional implicatures:

- in the old denotations  $\llbracket \text{SAID}_{\text{DS}} \rrbracket$ , we added the new accommodate handler to state that conventional implicatures should not project out of quoted speech
- in the new denotation  $\llbracket \text{APPOS} \rrbracket$ , we used the old SI handler to state that appositives should form scope islands

Otherwise, none of the denotations prescribed in our semantic lexicon had to be changed. We did not have to type-raise non-quantificational NP constructors like  $\llbracket \text{JOHN} \rrbracket$ ,  $\llbracket \text{ME} \rrbracket$  or  $\llbracket \text{BEST-FRIEND} \rrbracket$ . With the exception of direct speech, we did not have to modify any existing denotations to enable us to collect conventional implicatures from different subconstituents.

Furthermore, all of the modifications we have performed to existing denotations are additions of handlers for new effects. This gives us a strong guarantee that all of the old results are conserved, since applying a handler to a computation which does not use the operations being handled changes nothing.

The goal of our calculus is to enable the creation of semantic lexicons with a high degree of separation of concerns. In this section, we have seen how it can be done for one particular fragment.

## 4 Properties of the Calculus

The calculus defined in Sect. 2 and to which we will refer as  $(\lambda)$ , has some satisfying properties.

First of all, the reduction rules preserve types of terms (subject reduction). The reduction relation itself is confluent and, for well-typed terms, it is also terminating. This means that typed  $(\lambda)$  is strongly normalizing.

The proof of subject reduction is a mechanical proof by induction. For confluence and termination, we employ very similar strategies: we make use of general results and show how they apply to our calculus. Due to space limitations, we will pursue in detail only the proof of confluence.

Our reduction relation is given as a set of rules which map redexes matching some pattern into contracta built up out of the redexes' free variables. However, our language also features binding, and so some of the rules are conditioned on whether or not certain variables occur freely in parts of the redex. Fortunately, such rewriting systems have been thoroughly studied. Klop's Combinatory Reduction Systems (CRSs) [16] is one class of such rewriting systems.

We will make use of the result that orthogonal CRSs are confluent [16]. A CRS is *orthogonal* if it is left-linear and non-ambiguous. We will need to adapt our formulation of the reduction rules so that they form a CRS and we will need to check whether we satisfy left-linearity and non-ambiguity (we will see what these two properties mean when we get to them).

We refer the reader to [16] for the definition of CRSs. The key point is that in a CRS, the free variables which appear in the left-hand side of a rewrite rule, called metavariables, are explicitly annotated with the set of all free variables that are allowed to occur within a term which would instantiate them. This allows us to encode all of our  $x \notin FV(M)$  constraints.

One detail which must be taken care of is the set notation  $(\text{op}_i: M_i)_{i \in I}$  and the indices  $I$  used in the  $(\lambda)$  rules. We do away with this notation by adding a separate rule for every possible instantiation of the schema. This means that for each sequence of distinct operation symbols  $\text{op}_1, \dots, \text{op}_n$ , we end up with:

- a special rewriting rule  $(\text{op}_1: M_1, \dots, \text{op}_n: M_n, \eta: M_\eta) (\eta N) \rightarrow M_\eta N$
- for every  $1 \leq i \leq n$ , a special rewriting rule
 
$$(\text{op}_1: M_1, \dots, \text{op}_n: M_n, \eta: M_\eta) (\text{op}_i N_p (\lambda x. N_c(x)))$$

$$\rightarrow M_i N_p (\lambda x. (\text{op}_1: M_1, \dots, \text{op}_n: M_n, \eta: M_\eta) N_c(x))$$
- for every  $\text{op}' \in \mathcal{E} \setminus \{\text{op}_i | 1 \leq i \leq n\}$ , a special rewriting rule
 
$$(\text{op}_1: M_1, \dots, \text{op}_n: M_n, \eta: M_\eta) (\text{op}' N_p (\lambda x. N_c(x)))$$

$$\rightarrow \text{op}' N_p (\lambda x. (\text{op}_1: M_1, \dots, \text{op}_n: M_n, \eta: M_\eta) N_c(x))$$

So now we have a CRS which defines the same reduction relation as the rules we have shown in 2.3. Next, we verify the two conditions. Left-linearity states that no left-hand side of any rule contains multiple occurrences of the same metavariable. By examining our rules, we find that this is indeed the case.<sup>11</sup>

<sup>11</sup> Multiple occurrences of the same  $\text{op}_i$  are alright, since those are not metavariables.

Non-ambiguity demands that there is no non-trivial overlap between any of the left-hand sides.<sup>12</sup> In our CRS, we have overlaps between the  $\beta$  and the  $\eta$  rules. We split our CRS into one with just the  $\eta$  rule ( $\rightarrow_\eta$ ) and one with all the other rules ( $\rightarrow_{\langle\lambda\rangle}$ ). Now, there is no overlap in either of these CRSs, so they are both orthogonal and therefore confluent.

We then use the Lemma of Hindley-Rosen [17, p. 7] to show that the union of  $\rightarrow_{\langle\lambda\rangle}$  and  $\rightarrow_\eta$  is confluent when  $\rightarrow_{\langle\lambda\rangle}$  and  $\rightarrow_\eta$  are both confluent and commute together. For that, all that is left to prove is that  $\rightarrow_{\langle\lambda\rangle}$  and  $\rightarrow_\eta$  commute. Thanks to another result due to Hindley [17, p. 8], it is enough to prove that for all  $a, b$  and  $c$  such that  $a \rightarrow_{\langle\lambda\rangle} b$  and  $a \rightarrow_\eta c$ , we have a  $d$  such that  $b \rightarrow_\eta d$  and  $c \rightarrow_{\langle\lambda\rangle} d$ . The proof of this is a straightforward induction on the structure of  $a$ .

## 5 Conclusion

In our contribution, we have introduced a new calculus motivated by modelling detailed semantics and inspired by current work in programming language theory. Our calculus is an extension of the simply-typed lambda calculus which is the de facto lingua franca of semanticists. Its purpose is to facilitate the communication of semantic ideas without depending on complex programming languages [19,15] and to do so with a well-defined formal semantics.

We have demonstrated the features of our calculus on several examples exhibiting phenomena such as deixis, quantification and conventional implicature. While our calculus still requires us to do some uninteresting plumbing to be able to correctly connect all the denotations together, we have seen that the resulting denotations are very generic. We were able to add new phenomena without having to change much of what we have done before and the changes we have made arguably corresponded to places where the different phenomena interact.

Finally, we have also shown that the calculus shares some of the useful properties of the simply-typed lambda calculus, namely strong normalization.

In future work, it would be useful to automate some of the routine plumbing that we have to do in our terms. It will also be important to test the methodology on larger and more diverse fragments (besides this fragment, we have also created one combining anaphora, quantification and presupposition [19]). Last but not least, it would be interesting to delve deeper into the foundational differences between the approach used here, the monad transformers used by Charlow [4] and the applicative functors used by Kiselyov [13].

## References

1. Barker, C.: Continuations and the nature of quantification. *Natural language semantics* 10(3), 211–242 (2002)
2. Bauer, A., Pretnar, M.: Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.* 84(1), 108–123 (2015)

<sup>12</sup> The definition of (non-trivial) overlap is the same one as the one used when defining critical pairs. See [16] for the precise definition.

3. Brady, E.: Programming and reasoning with algebraic effects and dependent types. In: ACM SIGPLAN Notices (2013)
4. Charlow, S.: On the semantics of exceptional scope. Ph.D. thesis, New York University (2014)
5. de Groote, P.: Towards abstract categorial grammars. In: Proceedings of the 39th Annual Meeting on Association for Computational Linguistics (2001)
6. de Groote, P.: Type raising, continuations, and classical logic. In: Proceedings of the thirteenth Amsterdam Colloquium (2001)
7. de Groote, P.: Towards a montagovian account of dynamics. In: Proceedings of SALT. vol. 16 (2006)
8. Giorgolo, G., Asudeh, A.: Natural language semantics with enriched meanings (2015)
9. de Groote, P.: On Logical Relations and Conservativity. In: NLCS'15. Third Workshop on Natural Language and Computer Science (2015)
10. Hobbs, J., Rosenschein, S.: Making computational sense of montague's intensional logic. *Artificial Intelligence* 9(3), 287–306 (1977)
11. Kammar, O., Lindley, S., Oury, N.: Handlers in action. In: ACM SIGPLAN Notices (2013)
12. Kamp, H., Reyle, U.: From discourse to logic. Kluwer Academic Pub (1993)
13. Kiselyov, O.: Applicative abstract categorial grammars. In: Proceedings of the Third Workshop on Natural Language and Computer Science (2015)
14. Kiselyov, O., Sabry, A., Swords, C.: Extensible effects: an alternative to monad transformers. In: ACM SIGPLAN Notices (2013)
15. Kiselyov, O., Shan, C.c.: Lambda: the ultimate syntax-semantics interface (2010)
16. Klop, J.W., Van Oostrom, V., Van Raamsdonk, F.: Combinatory reduction systems: introduction and survey. *Theoretical computer science* (1993)
17. Klop, J.W., et al.: Term rewriting systems. *Handbook of logic in computer science* 2, 1–116 (1992)
18. Lebedeva, E.: Expression de la dynamique du discours à l'aide de continuations. Ph.D. thesis, Université de Lorraine (2012)
19. Maršík, J., Amblard, M.: Algebraic Effects and Handlers in Natural Language Interpretation. In: *Natural Language and Computer Science* (Jul 2014)
20. Meijer, E., Fokkinga, M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In: *Functional Programming Languages and Computer Architecture* (1991)
21. Moggi, E.: Notions of computation and monads. *Information and computation* 93(1), 55–92 (1991)
22. Plotkin, G., Pretnar, M.: Handlers of algebraic effects. In: *Programming Languages and Systems*, pp. 80–94. Springer (2009)
23. Potts, C.: The logic of conventional implicatures. Oxford University Press (2005)
24. Pretnar, M.: Logic and handling of algebraic effects. Ph.D. thesis, The University of Edinburgh (2010)
25. Qian, S., Amblard, M.: Event in compositional dynamic semantics. In: *Logical Aspects of Computational Linguistics*. Springer (2011)
26. Shan, C.: Monads for natural language semantics. arXiv cs/0205026 (2002)
27. Shan, C.: Linguistic side effects. Ph.D. thesis, Harvard University (2005)
28. Van Eijck, J., Unger, C.: Computational semantics with functional programming. Cambridge University Press (2010)
29. Wadler, P.: The essence of functional programming. In: *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1992)